Esqueletização de uma Imagem

Bruno Miguel da Silva Barbosa

Emanuel Queiroga Amorim Fernandes

Paradigmas de Computação Paralela

Mestrado Integrado em Engenharia Informática

Universidade do Minho, Departamento de Informática, Braga, Portugal

{a67646,a61003}@alunos.uminho.pt

Resumo. Neste relatório é apresentada a análise de um algoritmo de esqueletização de imagem no âmbito do paradigma de programação baseado em memória distribuída recorrendo à Interface de Passagem de Mensagens do *OpenMPI*. A abordagem do problema começa pela análise do algoritmo, para se perceber a forma de extração de paralelismo num sistema paralelo de memória distribuída, seguindo-se a sua implementação e para finalizar, a parte de experimentação que inclui, cálculos de tempos de execução em diversos nós do *cluster* e imagens de inputs diferentes, sendo concluindo o trabalho com análise dos resultados. Por último, apresentamos uma conclusão referente à distinção entre estas duas metodologias de programação. Indicamos também os dados de entrada e as otimizações realizadas. Nos anexos encontram-se as imagens resultantes (esqueletos) durante toda a experimentação, como algumas informações adicionais relevantes.

1. Introdução

A esqueletização de uma imagem é uma das inúmeras operações de processamento de imagem, com bastantes aplicações úteis para a sociedade como por exemplo: identificação de uma assinatura, reconhecimento de caracteres, reconhecimento de impressões digitais e ajudar no processamento de imagens relacionadas com a medicina. O algoritmo usado neste trabalho, varre todos os *pixéis* de uma imagem, e verifica se o pode remover através de cálculos com os *pixéis* adjacentes, podendo ter de percorrer a imagem mais do que uma vez. Portanto, percebemos instantaneamente, que se tivermos uma imagem com grande resolução, que é bastante comum nas câmaras fotográficas de hoje em dia, a execução do algoritmo pode ser muito ineficiente. Para reduzir o tempo de execução, a nossa abordagem visa uma implementação do mesmo algoritmo agora de forma paralela e distribuída através do uso da interface de passagem de mensagens *OpenMPI*. A paralelização torna-se deste modo, fundamental uma vez que permite o processamento de vários pixéis em simultâneo, tornando a esqueletização da imagem mais eficiente, uma condição vital quando são necessários resultados em tempo real, ou o algoritmo seja aplicado a grandes volumes de dados (neste caso imagens complexas de grande resolução ou muitas imagens diferentes). Para além da paralelização do algoritmo iremos também efetuar algumas otimizações com o objetivo de melhorar a sua performance.

2. O algoritmo

São feitas duas passagens pela imagem onde a cada iteração é realizado um conjunto de condições que determinam se um *pixel* devera pertencer, ou não, ao esqueleto da imagem. Para se remover o *pixel* todas as condições têm de ser respeitadas. Para um dado *pixel*, que não pertença à borda da imagem, temos a representação que se segue:
A nível de pseudocódigo, o algoritmo é descrito da seguinte maneira:
Repetir {
  1ª Passagem - Remover p1 se {

(i) O número de vizinhos a 1 está entre 2 e 6.
(ii) O número de transições 0-1 é igual a 1.
(iii) (p2 && p4 && p6) == 0 && (p4 && p6 && p8) == 0
}
2ª Passagem - Remover p1 se {
(i) O número de vizinhos a 1 está entre 2 e 6.
(ii) O número de transições 0-1 é igual a 1.
(iii) (p2 && p4 && p8) == 0 && (p2 && p6 && p8) == 0
}
} Até não remover mais pixéis

p9	p2	p3
p8	p1	p4
p7	p6	p5

Em termos de análise do código conseguimos identificar uma dependência de dados pois o valor do *pixel* atual é alterado tendo em conta os valores dos seus vizinhos. Esta dependência com pontos da vizinhança não existe quando: estes são "brancos", não vão ser alterados, ou quando não tem de remover, os restantes pontos não precisam deste para falhar a condição. Ou seja, traduzindo para código, temos um ciclo *while* que itera enquanto hajam alterações. Dentro deste temos um primeiro ciclo *for* que é responsável pela primeira passagem e pelas condições, através de *if*, a ela associadas. Depois temos ainda um segundo ciclo *for* responsável pela segunda passagem e que é muito idêntico ao primeiro alterando apenas na terceira condição *if*. Caso seja efetuada uma alteração (quando um pixel passa de 1 a 0) a variável que determina que houve alteração toma o valor 1.
Para extração de paralelismo de forma distribuída podemos dividir o domínio em N "fatias" horizontais da imagem, aproveitando assim a vantagem em memória dos dados estarem contínuos e haver no máximo duas "bordas" que precisem de comunicação. Não vamos adotar uma esquema em que tentamos calcular linhas que não tenham dependências das seguintes por o seguinte motivo: Para as imagens que vamos usar (letras gordas, para o qual o algoritmo produz um esqueleto satisfatório) apenas é executada uma iteração a remover pontos, onde um algoritmo para achar pontos sem dependências não seria vantajoso para reduzir o tempo de execução.
Como apenas os pontos a "preto" têm computação, o balanceamento pode ser feito, dando a cada *thread* um número aproximado destes pontos, esperando assim obter um tempo de execução semelhante entre as tarefas. Por este motivo, e para reduzir a diferença de imagem paralela em relação à sequencial (tendo o mínimo de divisões do domínio possível), vamos adotar um escalonamento estático do domínio.

3. Versão paralela

Relativamente à paralelização do algoritmo, o procedimento de distribuição de tarefas foi exatamente igual ao do trabalho anterior, em *OpenMP*. A imagem será dividida em segmentos iguais (inicialmente sem balanceamento), ou seja, com o mesmo número de linhas, e cada processo está responsável pela execução do segmento a ele respetivo.

A nossa implementação paralela seguiu o esquema de um algoritmo *master-slave*. Portanto existem dois tipos de processos: o master, que trata de coordenar os processos *slaves* através de primitivas de troca de mensagens, enviando as partes das matrizes dos restantes processos, o master faz algum balanceamento de carga, enviando um número díspar de linhas a cada trabalhador, mas com número de pontos a "preto" semelhantes; e os *slaves* que tratam de efetuar a esqueletização do segmento a eles atribuído e enviar de volta do bloco de matriz. Mais uma vez, persistem os habituais problemas de dependências de dados nas transições de um segmento para o outro. De modo a garantir a que os processos executam os mais simultaneamente possível, usamos umas barreiras entre os instantes em que são enviados/recebidos os segmentos das matrizes para cada processo.

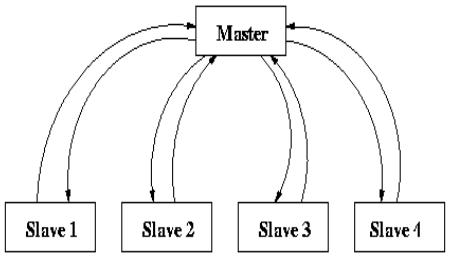


Figura 1. Algoritmo master-slave

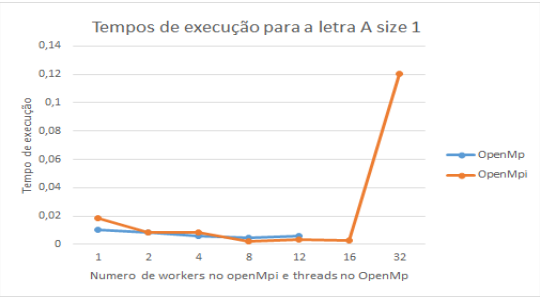
Nesta secção usamos os seguintes comandos
$ module load gnu/4.9.0
$ module load gnu/openmp_eth/1.8.4
$ mpicc -O3 ficheiro.c -o par -fopenmp
$ mpirun -mca btl tcp,sm,self --map-by {core,node} -np [2,32] par imagens/nome_da_imagem.ascii.pbm

4. Experimentação

Nesta secção apresentamos os resultados que consideramos mais relevantes tendo em conta os pontos de avaliação propostos. As imagens usadas têm as especificações como apresenta a tabela 1. Primeiramente, começamos por apresentar um gráfico que compara os tempos de execução entre os paradigmas de memória partilhada e memória distribuída tanto no modo sequencial como no modo paralelo. Foi usada a imagem letter_a_50.ascii.pbm com apenas 1 nodo (641) e mapeado por *core*.

Letter A	data size	side length
size 1	3.1 mb	1250
size 2	12.5 mb	2500
size 3	28.1 mb	3750

Tabela 1. Tamanhos das imagens



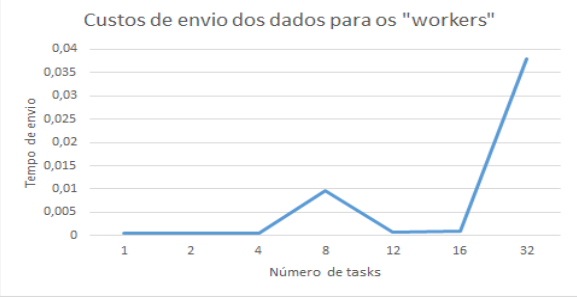
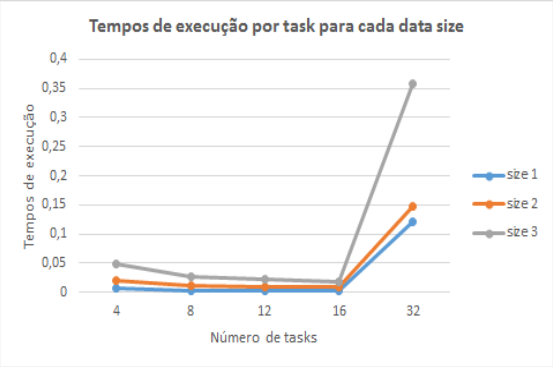
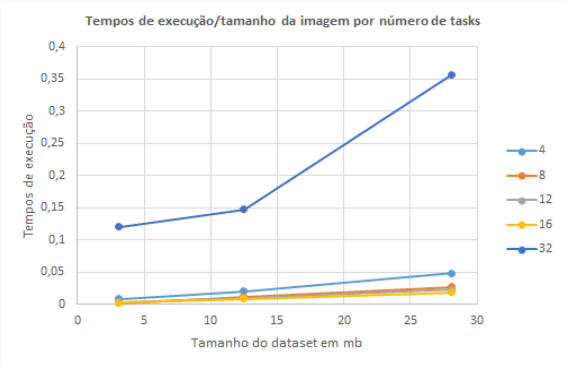


Figura 2. Tempos de execução e comunicação - openmp vs openmpi

De seguida, o nosso foco vira-se para o paradigma de memória distribuída fazendo uma análise ao nível da escalabilidade, dos custos de comunicação. Desta vez, as comparações serão feitas ao nível do tamanho da imagem de entrada. Ou seja, a comparação terá em conta as imagens com tamanhos 1, 2 e 3 (ver tabela 1). A razão desta escolha vem pelo facto de os nodos terem 20KB de LLC e querermos testar o impacto de ter que aceder a outro nodo. Note que, nesta fase ainda foi utilizado apenas um nodo (641) com o mapeamento por *cores*.





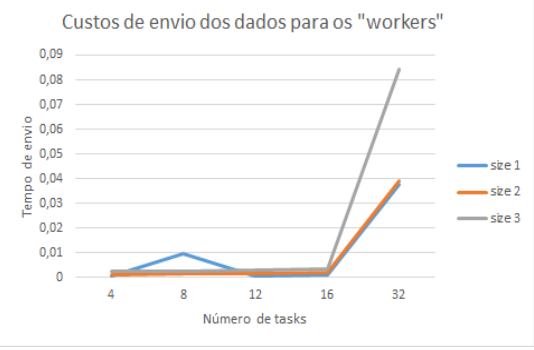


Figura 3. Tempos de execução por tamanho

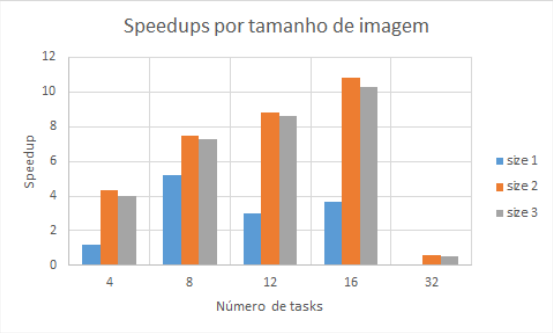


Figura 4.Speedups

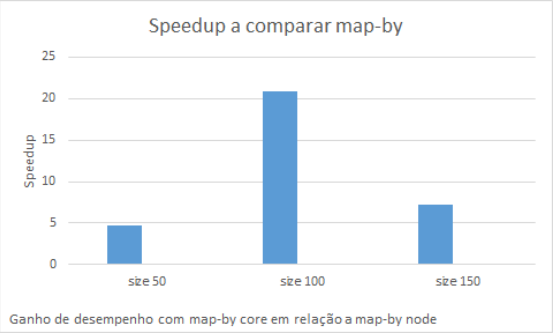


Figura 5. Comparação map-by

Na figura 4 utilizamos um nodo (641) com mapeamento por core. Contudo, na figura 5 já comparamos, para 8 *tasks*, a influência de cada tipo de mapeamento em 2 nodos (641). Nos anexos, existe uma secção que continua com os gráficos correspondentes à experimentação.

5. Análise de resultados

Pela análise dos resultados da experimentação verificamos o seguinte:

- Pela figura 2 conseguimos verificar que quando executamos sequencialmente, o paradigma da memória partilhada executa em menor tempo de onde podemos inferir que tem menos custos de *overhead*. À medida que aumentamos o número de *tasks* apuramos que a versão do MPI torna-se gradualmente mais eficiente, no entanto os custos de comunicação tornam-se cada vez mais custosos, como podemos observar quando temos 32 *tasks*. Uma das possíveis causas da versão em *OpenMPI* estar a ser mais eficiente pode ser devido ao balanceamento de carga realizado. (nota: na versão de *OpenMP* só foram realizados testes até 12 *tasks* porque já era notável a perda de escalabilidade.)
- Através da figura 3 conseguimos identificar que o tempo de execução é proporcional ao aumento de tamanho da imagem destacando em especial, o caso das 32 *tasks* quando o tamanho é máximo, onde se nota a grande diferença entre os tamanhos anteriores. Conjugando os gráficos de custo de comunicação e de tempos de execução para cada tamanho de imagem concluímos que o quanto maior for o tamanho dos dados maior será o tempo gasto em comunicação.
- A figura 4 corresponde aos *speedups* obtidos comparando a versão sequencial à implementação da versão paralela do algoritmo de esqueletização. Como seria esperado, podemos notar de imediato que para 32 *tasks* o algoritmo não escala, obtendo tempos de execução superiores ao da versão sequencial. Contudo, no geral, o *speedup* aumenta de acordo com o número de *tasks* sendo cada vez menos relevante. Verifica-se que o incremento mais significativo do *speedup* ocorre entre 4 e 8 *tasks*. Em relação aos tamanhos das imagens, o algoritmo melhora a performance até ao size 2, onde praticamente estagna.
- Na figura 5 temos a análise correspondente ao mapeamento por *cores* e por nodos. O gráfico diz-nos que o mapeamento por core é mais favorável quando os dados não preenchem uniformemente as memórias cache de cada nodo. Ou seja, no nosso caso, cada nodo tem uma LLC de 20MB enquanto que o maior tamanho da nossa imagem chega apenas a, aproximadamente, 28MB.
- A figura 9 (nos anexos) demonstra o contraste dos tempos de execução quando se usam 1 ou 2 nodos. Pela sua averiguação concluímos que para o nosso algoritmo e para os dados usados, não compensa usar 2 nodos.
- A última análise foca-se em averiguar o contraste nos tempos de execução em 2 máquinas distintas. Por questões de simplicidade, apresentamos os resultados apenas para 1 nodo de cada máquina (641 e 662), com 8 *tasks* e para os diferentes tamanhos de imagem recorrendo ao mapeamento por core. A figura 10 (nos anexos) demonstra-nos que à máquina 641 é mais eficiente para todos os tamanhos.
- As imagens resultantes da esqueletização podem ser consultadas nos anexos.

Anexos

Dados de entrada

Como dados de entrada utilizamos duas imagens com a extensão *.ascii.pbm*. As imagens tiveram que ser aumentadas visto que, inicialmente, os seus tamanhos não excediam os 25KB. Neste sentido, criamos um programa que tratava de fazer o *resize* das mesmas mantendo a sua proporção, de modo a que estas possuíssem um tamanho na ordem dos MB. Nos anexos encontram-se as imagens utilizadas e o código correspondente ao programa do *resize*.

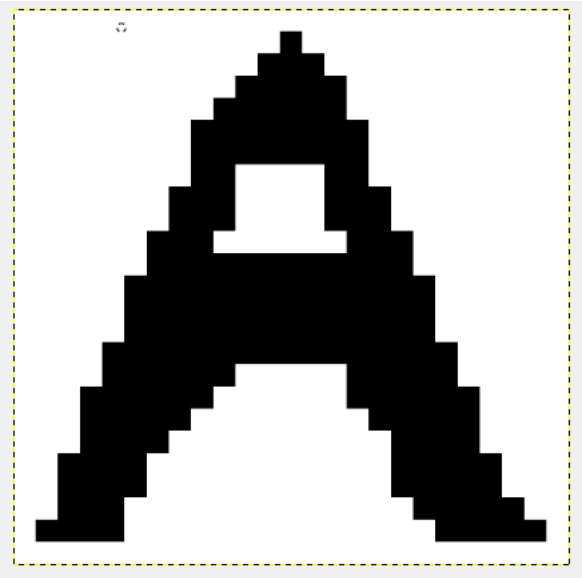


Figura 6. Dados de entrada (já com rescaling)

Código resize

```
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<unistd.h>
#include<ctype.h>

#define ROW 5000         //define o máximo de linhas
#define COL 5000         //define o máximo de colunas
#define MULT 150
char tipo[3];           //tipo de ficheiro
int mat[ROW][COL];      //matriz onde será guardada a imagem
int linhas,colunas;     //garantir que linha < ROW e coluna < COL
                        //#linhas e #colunas efectivamente usadas

/**
 * Carrega imagem do tipo .pbm
 * @param path
 * @return
 */


/**
 * Cria a imagem do esqueleto
 */
void imprimeMatriz() {
    int i,j;
    FILE *fp;

    fp = fopen("letter_a_150.ascii.pbm","w");
    fprintf(fp,"%s\n",tipo);
```

```
    fprintf(fp,"%d %d\n",colunas,linhas);

    for(i=0;i<linhas;i++) {
        for(j=0;j<colunas;j++) {
            fprintf(fp,"%d ",mat[i][j]);
        }
        fprintf(fp,"\n");
    }
    fclose(fp);
}

int carregaImagemPBM(char *path) {
    FILE *fp;
    //char tipo[3];
    int l,c,i,j;

    fp = fopen(path,"r");

    if(fp) {
        fscanf(fp,"%s",tipo);     //ok
        fscanf(fp,"%d %d",&c,&l); //ok
        linhas = l;
        colunas = c;
       int mult = MULT;
       int i2=0,j2=0;
       int ci=0, cj=0;
       int act=0;
        //printf("Tipo %s\nLinhas %d Colunas %d\n",tipo,l,c);

        for(i=0;i<l;i++) {
            for(j=0;j<c;j++) {
                fscanf(fp,"%d",&act);
             for(ci=0;ci<mult;ci++){
                    for(cj=0;cj<mult;cj++){
                    mat[i2][j2]=act;
                    //printf("[%d , %d]",i2,j2);
                    j2++;
                    }
                    //printf("\n");
                    if(ci<mult-1){
                           j2=(j2-MULT);
                           i2++;
                    }else{
                           i2=i2-(MULT-1);
                           }
             }

            }
       i2+=MULT;
       j2=0;

        }
    }
    fclose(fp);
    linhas = linhas * MULT;
    colunas = colunas * MULT;
       imprimeMatriz();
    return 1;
}
/**
 * Calcula o complementar de um bit
 * @param num
 * @return
 */
int comp(int num) {
    if(num==0)
       return 1;
    else
```

```
       return 0;

}

/**
 * Verifica se houve transicao
 * @param act
 * @param ant
 * @return
 */
int trans(int act, int ant) {
  return (act == 1 && ant == 0);
}

int getValue(int i, int j){
int res=-1;

if(i<0 || i>(linhas-1) || j<0 || j>(colunas-1))
{
res=0;
}else{

res = mat[i][j];
}

return res;
}

int main(int argc, char **argv) {
    int alterou=1,i,j,vizinhos,transicoes,complementos;
    int p1, p2,p3,p4,p5,p6,p7,p8,p9;

    /* le ficheiro de input - verificar nargs */
    if(argc != 2) {
        alterou = 0;
        printf("#ERRO: Insira uma imagem\n");
    } else {
        carregaImagemPBM(argv[1]);
    }
}
```

Esqueletos

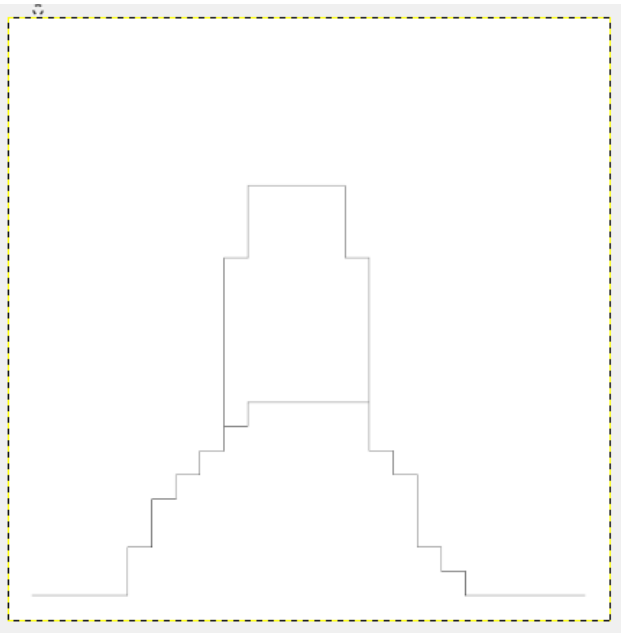


Figura 7. Versão sequencial (openmp e mpi)

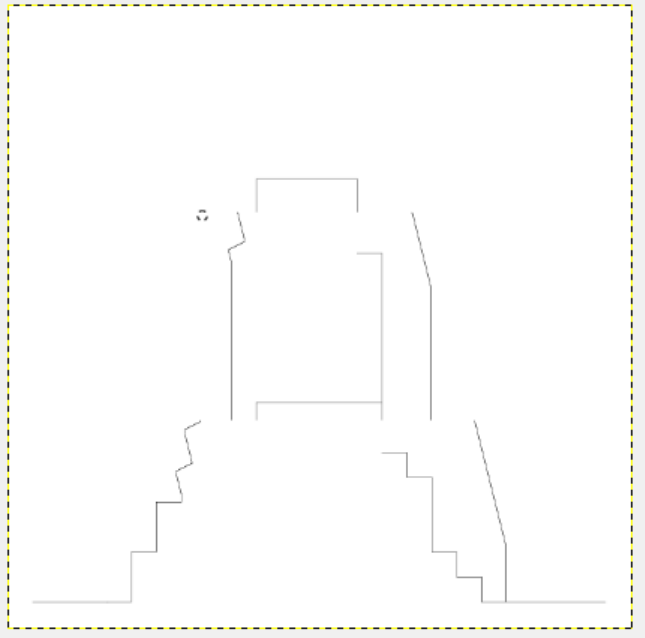


Figura 8. Versão openmpi com 3 processos (master + 2 slaves)

Nota: Os restantes esqueletos não possuem grande qualidade pelo que não os apresentamos.

Análise de resultados (continuação)

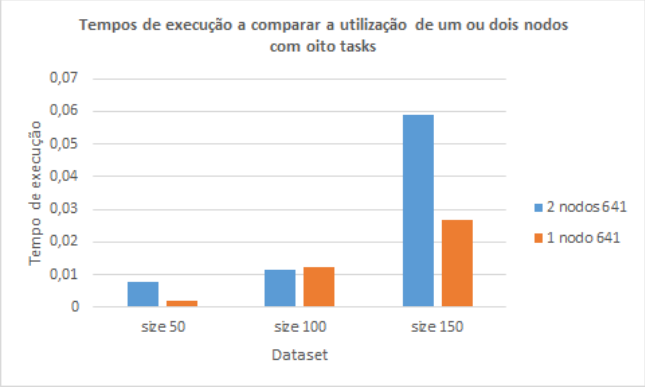


Figura 9. Tempos de execução variando número de nodos

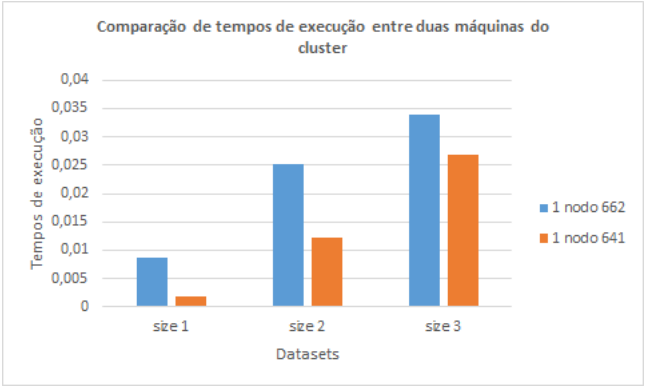


Figura 10. Tempos de execução em 2 máquinas